

# $\lambda$ -Calculus

Mattox Beckman

`beckman@iit.edu`

Illinois Institute of Technology

---

Subtitle: “Everything is Really a Function”

---

The purposes of this lecture is to demonstrate how functions themselves can represent arbitrary data structures.

- Know the three constructs of  $\lambda$ -calculus.
- Know the competitive advantage of  $\lambda$ -calculus.
- Know how to use functions to represent integers.
  - Know how to write addition, multiplication, and exponentiation.
- Know how to use functions to represent arbitrary types.
  - booleans
  - pairs

- Contains three kinds of things:
  1. Variables:  $x$
  2. Functions:  $\lambda x.x$
  3. Application:  $(\lambda x.x) y$... and nothing else!
- “We don’t need no stinking integers!”
- This language is Turing Complete.
- Used extensively in research. The “little white mouse” of computer science.
- We can implement this very easily in OCaml.  
 $\lambda x.x = \text{fun } x \rightarrow x$

The semantics of  $\lambda$ -calculus are very easy. Remember that  $M, N, P$  are expressions,  $V$  is a value.

**Variables**  $\frac{}{x \Downarrow x}$

**Functions**  $\frac{}{\lambda x.M \Downarrow \lambda x.M}$

**Application**  $\frac{M \Downarrow \lambda x.P \quad N \Downarrow V}{M N \Downarrow [V/x]P}$

What can you tell about this language?

Here's another version of λ-calculus.

**Variables**  $\frac{}{x \Downarrow x}$

**Functions**  $\frac{}{\lambda x.M \Downarrow \lambda x.M}$

**Application**  $\frac{M \Downarrow \lambda x.P}{M N \Downarrow [N/x]P}$

What is different about this version of the language?

We will add one “rule” to these:  $\frac{M \Downarrow \lambda x.N \quad N \Downarrow V}{M \Downarrow \lambda x.V}$

This is known as *reduction*, as opposed to *evaluation*. Can you see why this rule is not valid?

- The Lambda Calculus doesn't have numbers.
- A number  $n$  can be thought of as a potential: someday we are going to do something  $n$  times.

$f0 = \text{fun } f \ x \rightarrow x$

$f1 = \text{fun } f \ x \rightarrow f \ x$

$f2 = \text{fun } f \ x \rightarrow f \ (f \ x)$

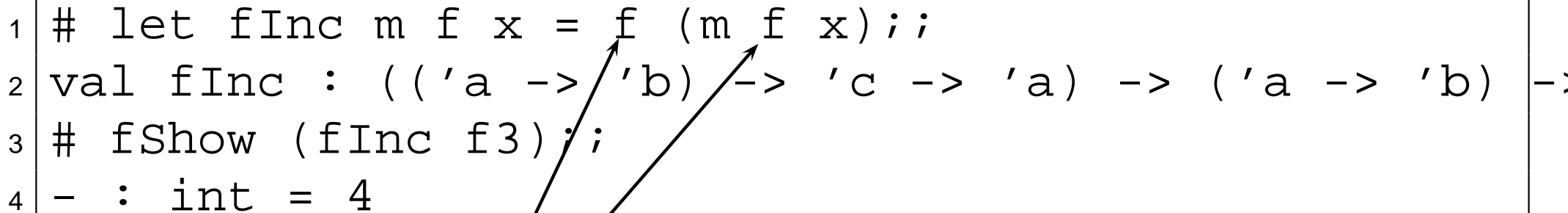
$f3 = \text{fun } f \ x \rightarrow f \ (f \ (f \ x))$

A numeral  $n$  takes a function and a value, and applies the function  $n$  times to the value.

```
1 # let fShow m = m ( (+) 1 ) 0;;
2 val fShow : ((int -> int) -> int -> 'a) -> 'a = <fun
3 # fShow f3;;
4 - : int = 3
```

- To increment a Church Numeral...

```
1 # let fInc m f x = f (m f x) ; ;  
2 val fInc : (('a -> 'b) -> 'c -> 'a) -> ('a -> 'b) -> 'c  
3 # fShow (fInc f3) ; ;  
4 - : int = 4
```



- First, apply  $f$   $m$  times to  $x$ .
- Next, apply  $f$  once more to the result.

- Similar reasoning can yield addition and multiplication.

```
1 # let fAdd m n f x = m f (n f x) ;;
2 val fAdd : ('a -> 'b -> 'c) ->
3           ('a -> 'd -> 'b) -> 'a -> 'd -> 'c = <fun>
4 # fShow (fAdd f3 f3) ;;
5 - : int = 6
6 # let fMul m n f x = m (n f) x ;;
7 val fMul : ('a -> 'b -> 'c) ->
8           ('d -> 'a) -> 'd -> 'b -> 'c = <fun>
9 # fShow (fMul f3 f3) ;;
10 - : int = 9
```

- Subtraction is *much* more complex.



```
1 # let fAdd m n f x = m f (n f x) ; ;  
2 # fAdd f2 f3 ; ;  
3 → f2 f (f3 f x)  
4 → f2 f ((fun g y -> g (g (g y))) f x)  
5 → f2 f (f (f (f x)))  
6 → (fun g y -> g (g y)) f (f (f (f x)))  
7 → (fun y -> f (f y)) (f (f (f x)))  
8 → f (f (f (f (f x))))
```

```

1 # let fMul m n f x = m (n f) x;;
2 # fMul f3 f3;;
3 → f3 (f3 f) x
4 → f3 ((fun g y -> g (g (g y))) f) x
5 → f3 (fun y -> f (f (f y))) x
6 → (fun h z -> h (h z)) (fun y -> f (f (f y))) x
7 → (fun y -> f (f (f y))) ((fun y -> f (f (f y)))
8                               ((fun y -> f (f (f y))) x
9 → (fun y -> f (f (f y))) ((fun y -> f (f (f y)))
10                               (f (f (f x))))
11 → (fun y -> f (f (f y))) (f (f (f (f (f (f x))))))
12 → (f (f (f (f (f (f (f (f (f x))))))))

```

Suppose we have a type  $S$ , with constructors  $S_1 \dots S_n$ . We can represent a term  $t = S_i(t_1, t_2, \dots, t_m)$  by

$$\bar{t} = \lambda x_1 \dots x_n. x_i \bar{t}_1 \dots \bar{t}_m$$

The  $x_i$  represent functions that say what to do if the term  $t$  turns out to be an instance of the constructor  $S_i$ .

Let  $S_1 = \text{true}$ , and  $S_0 = \text{false}$ .

Then, a boolean is a value which takes two functions. The first function is what to do when the boolean is `true`, the second is what to do when the boolean is `false`.

```
1 # let fTrue a b = a;;  
2 val fTrue : 'a -> 'b -> 'a = <fun>  
3 # let fFalse a b = b;;  
4 val fFalse : 'a -> 'b -> 'b = <fun>  
5 # let fIf c t e = c t e;;  
6 val fIf : ('a -> 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
```

● How would you define `fAnd` and `fOr` with this system?

A pair is a type with one constructor  $S_0 = \text{pair}$ . We know that  $S_0$  takes two arguments.

Then, a pair is a value that takes one function. The function itself takes two arguments, and tells what to do with the two halves of the pair.

```
1 # let fPair a b = fun select -> select a b;;  
2 val fPair : 'a -> 'b -> ('a -> 'b -> 'c) -> 'c = <fu  
3 # let fPi1 p = p (fun a b -> a);;  
4 val fPi1 : (('a -> 'b -> 'a) -> 'c) -> 'c = <fun>  
5 # let fPi2 p = p (fun a b -> b);;  
6 val fPi2 : (('a -> 'b -> 'b) -> 'c) -> 'c = <fun>
```

- You can combine these techniques to create arbitrary data structures.
- A list contains two constructors: `Cons` and `Nil`.
- The `Cons` constructor contains a pair as data.

```
1 # let fNil cons nil = nil;;  
2 val fNil : 'a -> 'b -> 'b = <fun>  
3 # let fCons a b cons nil = cons a b;;  
4 val fCons : 'a -> 'b -> ((( 'a -> 'b -> 'c) -> 'c) -> 'c) -> 'c  
5 # let fIsNil lst = lst (fun x y -> fFalse) fTrue;;  
6 val fIsNil : (('a -> 'b -> 'b) -> ('c -> 'd -> 'c) -> 'c) -> bool
```

You will need to tell OCaml to use recursive types for this... but even then the type system starts to get in the way.

1. (126) Write a function `church n` which returns the Church Numeral for  $n$ . (Use recursion.)
2. (127) Write the church function for exponentiation. (Hint: the solution is shorter than multiplication.) Show a sample run for  $3^2$ .
3. (128) How would you represent a tree using functions? (There are many ways to do this.)

- Write a function `church n` which returns the Church Numeral for  $n$ . (Use recursion.)

```
1 # let rec church n f x =  
2   match n with  
3     0 -> x  
4     | _ -> f (church (n-1) f x);;  
5 val church : int -> ('a -> 'a) -> 'a -> 'a = <fun>
```



- Write the church function for exponentiation. (Hint: the solution is shorter than multiplication.) Show a sample run for  $3^2$ .

Solution:

```
1 # let fPow m n = n m ;;
2 # fPow f3 f2
3 → f2 f3
4 → (fun f x -> f (f x)) f3
5 → fun x -> f3 (f3 x)
6 → fun x -> (fun g y -> g (g (g y))) (f3 x)
7 → fun x -> fun y -> (f3 x) ( (f3 x) ( (f3 x) y))
8 → fun x y -> (f3 x) ( (f3 x) (x (x (x y))))
9 → fun x y -> (f3 x) (x (x (x (x (x (x y))))))
10 → fun x y -> (x (x (x (x (x (x (x (x (x y))))))))
```

- How would you represent a tree using functions? (There are many ways to do this.)

```
1 # let mkLeaf n node leaf = leaf n;;  
2 val mkLeaf : 'a -> 'b -> ('a -> 'c) -> 'c = <fun>  
3 # let mkNode a b node leaf = node a b;;  
4 val mkNode : 'a -> 'b -> ('a -> 'b -> 'c) -> 'd ->
```

- You can use  $\lambda$ -calculus to represent itself using these techniques. You already have everything you need to do it. You can see the details in Torben  $\text{\AA}$ . Mogensen's paper Efficient Self-Interpretations in lambda Calculus, in the Journal of Functional Programming v2 n3.
- More information about using functions to represent objects can be found in J. Steensgaard-Madsen's paper Typed representation of objects by functions, in TOPLAS v11 n1.